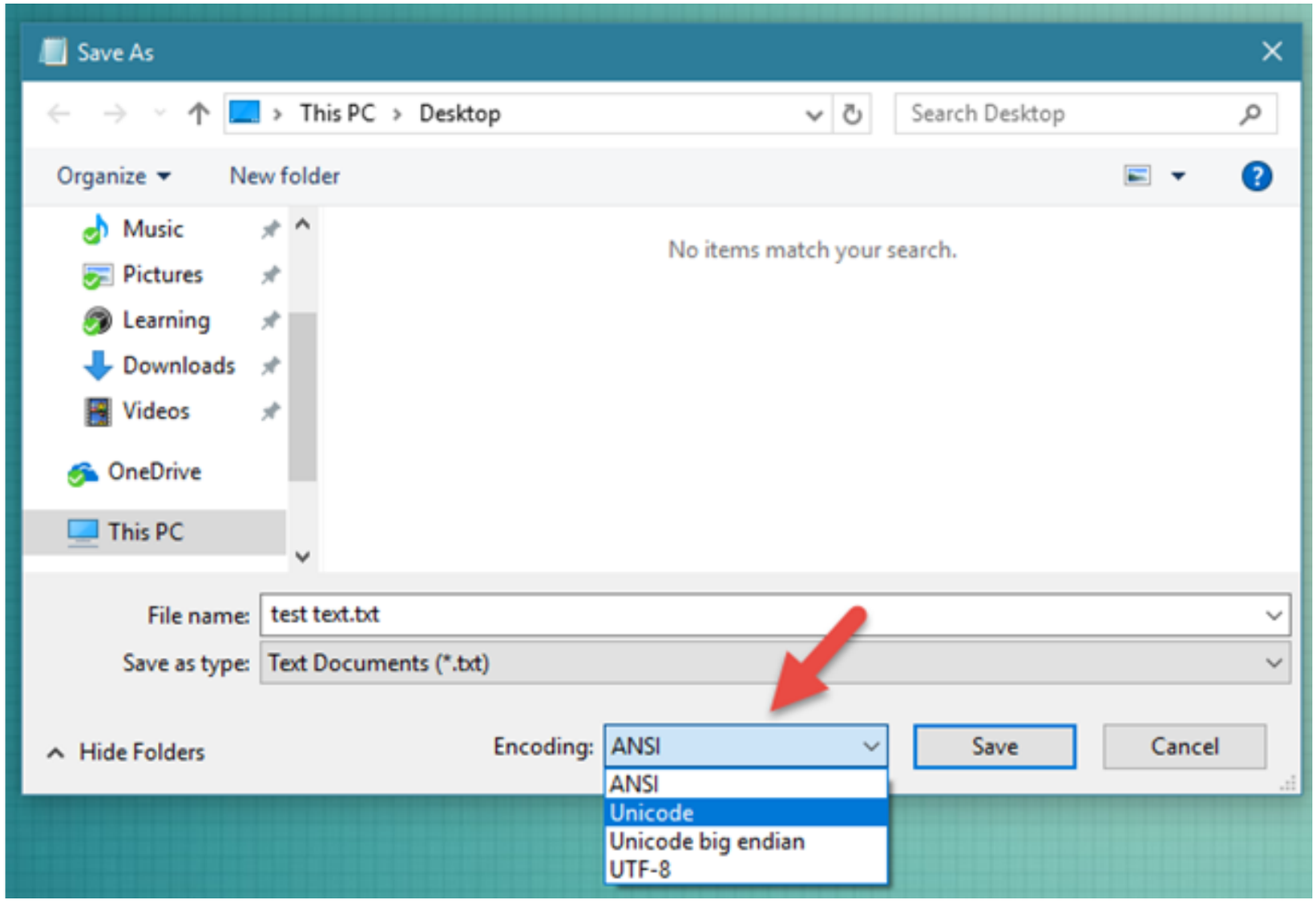
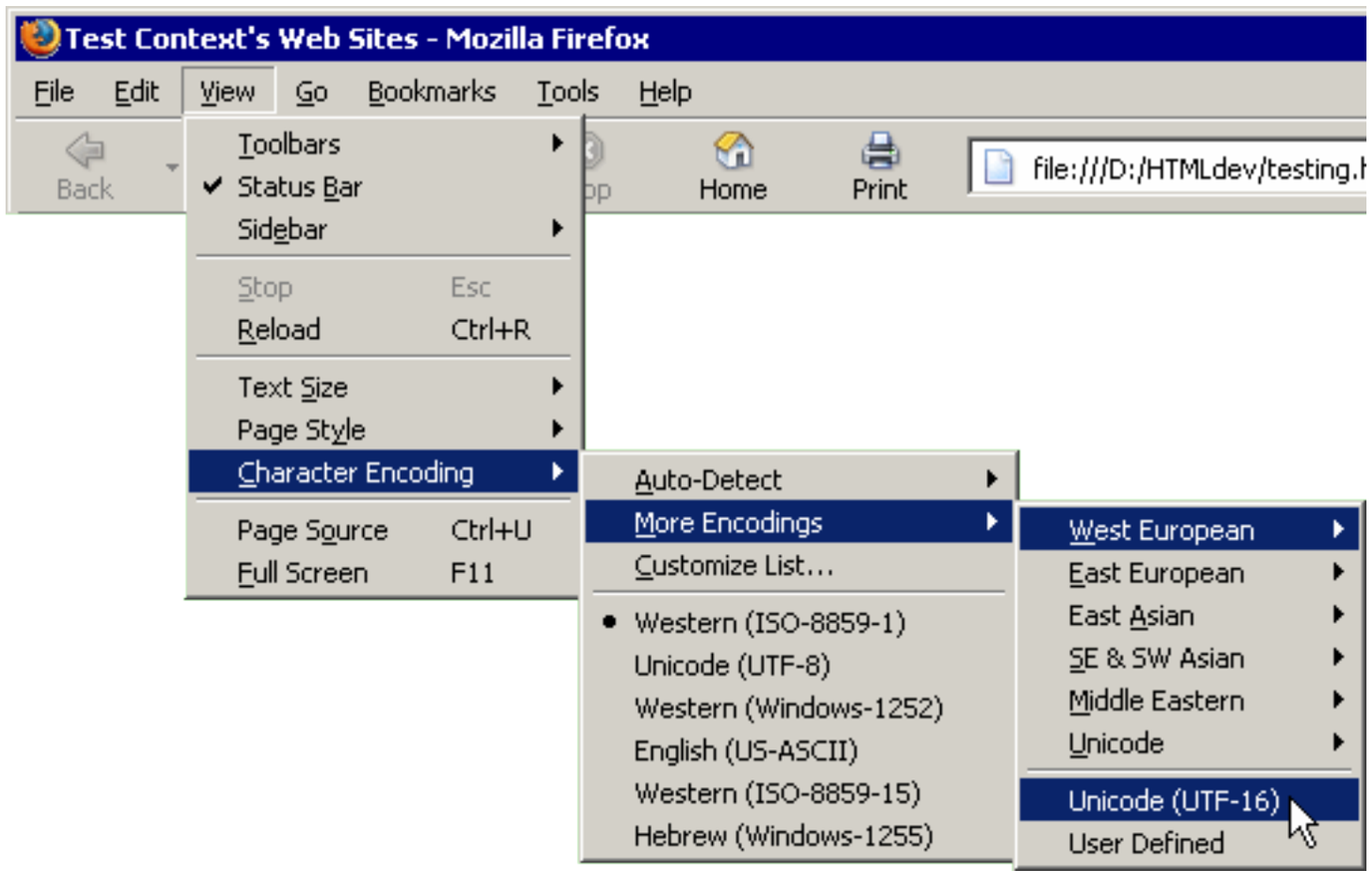
Unicode

Encoding Forms

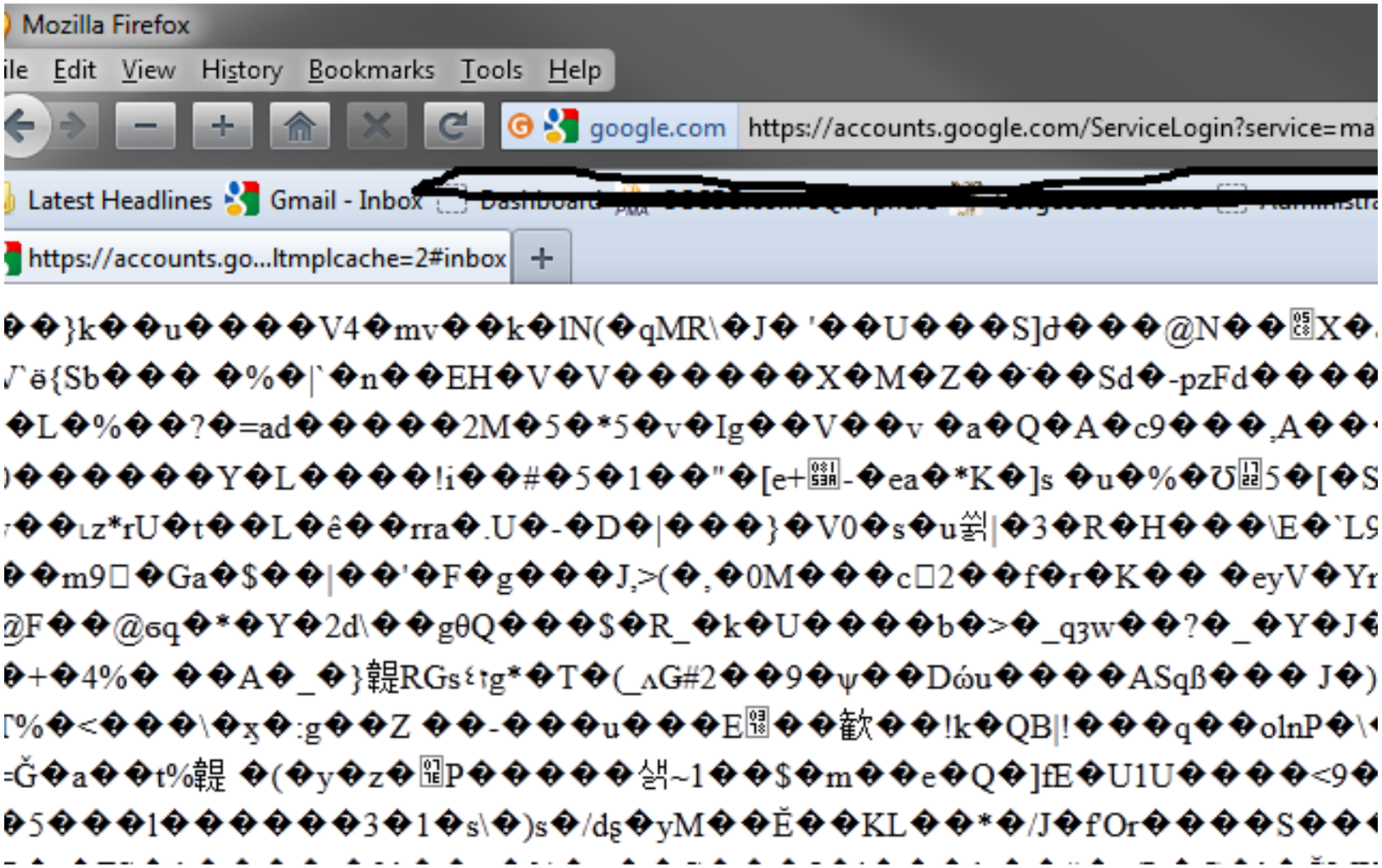
Which one do I choose? 🤔



Why are there so many? 🤔



What happened to the text? 🤔



Let’s start from the beginning

Once upon a time, when life was simpler (for Americans), we had ASCII

ASCII

- Designed for teleprinters in the 1960s
- 7 bit (0000000 to 1111111 in binary)
- **128** codes (0 to 127 in decimal)

A	0	1	0	0	0	0	0	1	*(65 in decimal)*
B	0	1	0	0	0	0	1	0	*(66 in decimal)*
C	0	1	0	0	0	0	1	1	*(67 in decimal)*

Meanwhile, 8-bit byte were becoming common

🤔

A	0	1	0	0	0	0	0	1	*(65 in decimal)*
B	0	1	0	0	0	0	1	0	*(66 in decimal)*
C	0	1	0	0	0	0	1	1	*(67 in decimal)*

Why waste this 1 bit?

(aka,128 more spaces for characters)

Everyone had the same idea-

Let’s Extend ASCII

And use all those ***256*** characters

Let there be… chaos! 😈

Code Pages

- There are too many (more than 220 DOS and Windows code pages alone)
- IBM, Apple all introduced their own “code pages” 👉 👈
- Side-note: **ANSI** character set has no well-defined meaning, they are a collection of 8-bit character sets compatible with ASCII but incompatible with each other

Problems

- Most are compatible with ASCII but incompatible with each other
- Programs need to know what code page to use in order to display the contents correctly
- Files created on one machine may be unreadable on another
- Even 256 characters are not enough 🙅🏻‍♂️

And also

Internet happened!

Fax machines were not enough anymore.

Unicode

to the rescue

Unicode

- Finally everyone agreed on what code point mapped to what character 🎉
- There's room for **over 1 million code points** (characters), though the majority of common languages fit into the first 65536 code points

How do we serialize multi-byte characters?

“Character Encoding”

UTF-32

- Simplest encoding
- Unicode supports 1,114,112 code points. We can store them using 21 bits
- UTF-32 is 32-bit in length. Fixed length encoding
- So we take a 21-bit value and simply zero-pad the value out to 32 bits

UTF-32

- Example
  A in ASCII 01000001
  A in UTF-32 00000000 00000000 00000000 01000001
  or, 01000001 00000000 00000000 00000000

- So, not compatible with ASCII

- Super wasteful

- But faster text operations (e.g character count)

- Messes where “null terminated strings” (00000000) are expected

UTF-32

- Also, now we have to deal with “**Endianness**” 🤔

00000000 00000000 00000000 01000001

vs,

01000001 00000000 00000000 00000000

Endianness

- Big endian machine: Stores data big-end first. When looking at multiple bytes, the first byte is the biggest

  increasing addresses — — —->
  00000000 00000000 00000000 01000001

- Little endian machine: Stores data little-end first. When looking at multiple bytes, the first byte is smallest

  increasing addresses — — —->
  01000001 00000000 00000000 00000000

- Endianness does not matter if you have a single byte, because how we read a single byte is same in all machines 😌

UTF-16

- The oldest encoding for Unicode. Often mislabeled as "Unicode encoding”

- Variable length encoding. 2 bytes for most common characters (BMP), 4 bytes for everything else

- The most common characters (BMP) in Unicode fits into first 65,536 code points, so it’s straightforward. Throw away top 5 zeros from 21 bit, you get UTF-16.
  A 00000 00000000 01000001 (21 bit) becomes-
  A 00000000 01000001 (16 bit)

- Uses “Surrogate pairs” for other characters

UTF-16

- Multi-byte encoding, so has Endianness like UTF-32

- Incompatible with ASCII
  A in UTF-16 00000000 01000001
  A in ASCII 01000001

- Incompatible with old systems that rely on null (null byte: 00000000) terminated strings

- Uses less space than UTF-32 in practice

- Windows API, .NET and Java environments are founded on UTF-16, often called “wide character string”

UTF-8

- Nice & simple. This is the kid that everyone loves *
- Backward compatible with ASCII 🙌
- 0 to 127 code points are stored as regular, single-byte ASCII.

  A in ASCII 01000001
  A in UTF-8 01000001

** UTF-8 Everywhere Manifesto: https://utf8everywhere.org/*

UTF-8

- Code points 128 and above are converted to binary and stored (encoded) in a series of bytes

A 2-byte example looks like this

110xxxxx	**10xxxxxx**
Count byte	**Data byte**
Starts with 11..0	Starts with 10

In contrast, single byte ASCII characters (<128 decimal code points) look like ***0xxxxxxx***

UTF-8

- **A 2-byte example looks like this**
  110xxxxx 10xxxxxx
  (Count Byte) (Data Byte)

- The first count byte indicates the number of bytes for the code-point, including the count byte. These bytes start with **11..0**:
  110xxxxx (The leading “11” is indicates 2 bytes in sequence, including the “count” byte)

  1110xxxx (1110 -> 3 bytes in sequence)

  11110xxx (11110 -> 4 bytes in sequence)

- After count bytes, data bytes starting with **10…** and contain information for the code point

UTF-8

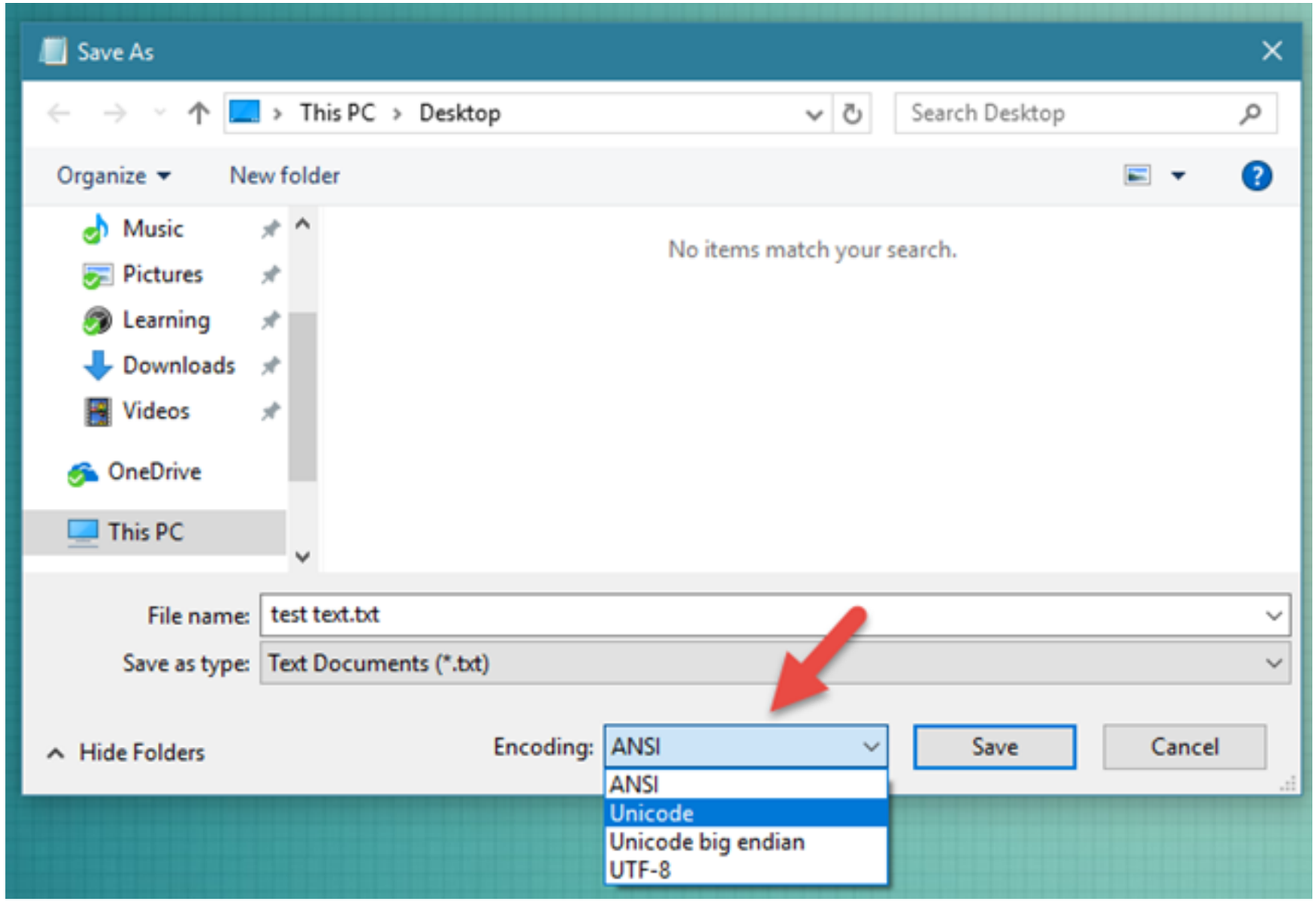
- Example:
  অ 'BENGALI LETTER A' (U+0985)
  Binary form: 00001001 10000101
  0000 100110 000101
  UTF-8 form: 11100000 10100110 10000101

- Variable length encoding. Uses more space than UTF-16 for text with mostly Asian characters (2 bytes vs 3 bytes), less space than UTF-16 for text with mostly ASCII characters (1 byte vs 2 bytes)

UTF-8

- No null bytes. Old programs work just fine that treats null bytes (00000000) as end of string
- We read and write a single byte at a time, so no worry of Endianness. This is very convenient
- UTF-8 is the encoding you should use if you work on web

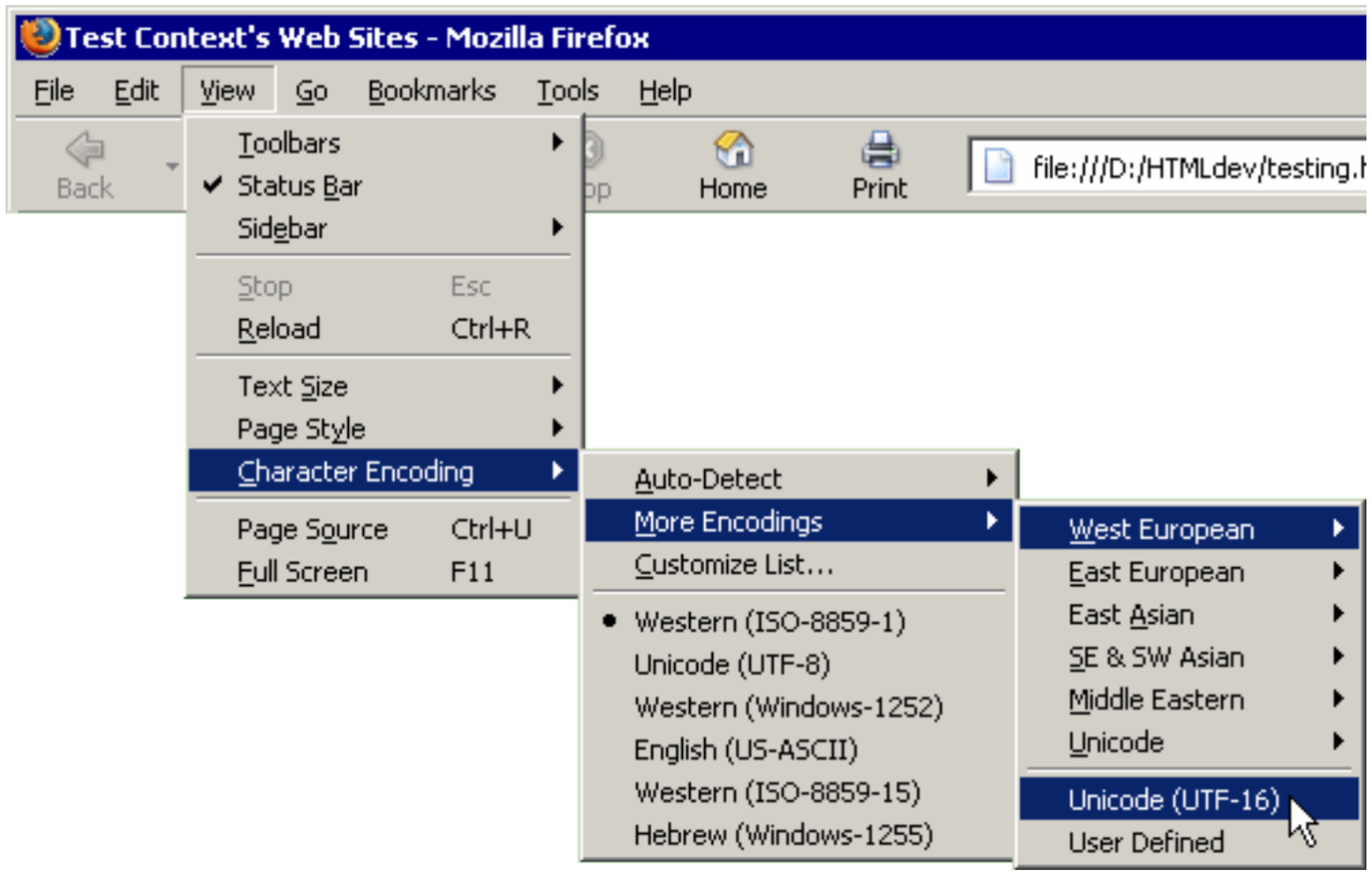
How does Notepad know which encoding was used when it opens a file?



BOM
(Byte Order Mask)

BOM	Encoding
00 00 FE FF	UTF-32, big-endian
FF FE 00 00	UTF-32, little-endian
FE FF	UTF-16, big-endian
FF FE	UTF-16, little-endian
EF BB BF	UTF-8

How do browsers know which encoding was used when it opens a page?



HTTP Content-Type Header

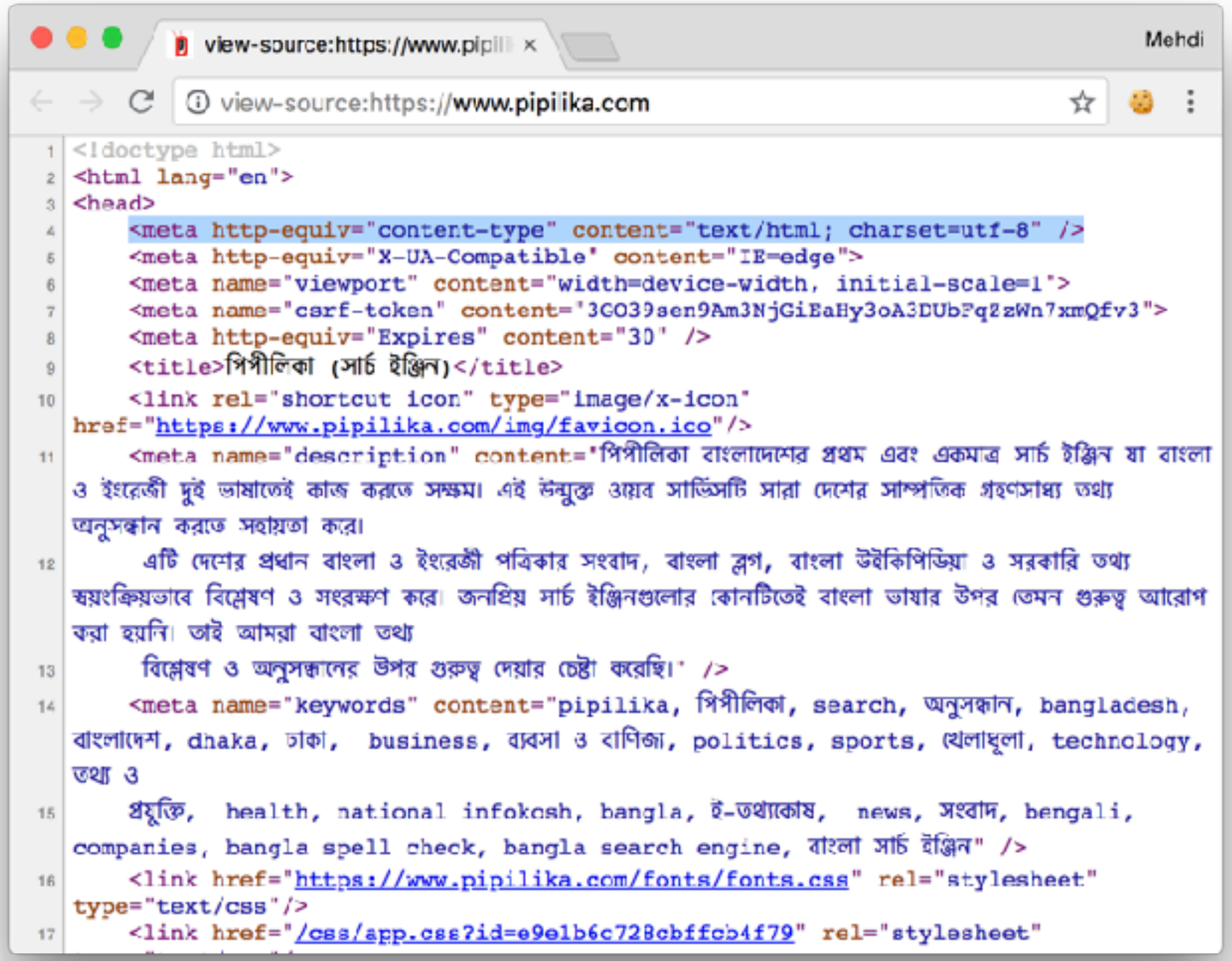
2. fish /Users/mehdi (fish)

fish /Users/mehdi (fi... ⌘1

```
~ curl -I https://www.pipilika.com/
HTTP/1.1 200 OK
Date: Wed, 25 Jul 2018 09:44:47 GMT
Server: Apache/2.4.18 (Ubuntu)
Cache-Control: no-cache, private
Set-Cookie: XSRF-TOKEN=eyJpdiI6IkkzRG9LWVFaOFpwWWxQekJxWWNyQmc9PSIsInZhbHVlIjoiSU9
DZXQxRzIrUzF5MkNVY2JtYlJTOTZVQjZcL0NtcDB1dkFxTitHOHhZZ0dMbFJoK3JETmdzRStkd0Y5SWR1S
mNqWFdZZ1E4ZTJRNm1kQVJXXC8yQUNudz09IiwibWFjIjoiZTZiOWJmZmYxZGJhZDIwMGE2NTM0YjY5NzA
xOTcyYmI2ZDI0ZWM1ZjhmM2NhZWYwYTljNjg3MDM1ODI4N2YyMCJ9; expires=Wed, 25-Jul-2018 11
:44:47 GMT; Max-Age=7200; path=/
Set-Cookie: pipilika_main_search_session=eyJpdiI6InBmeFBUZHdZSTdUbThLUkRxczNEZUE9P
SIsInZhbHVlIjoiOCtzRTlHd0YrZ0o2XC95OUdQc2JSa3pSNitwVldVd1dkVG9pa1Q5cDBpaXR1bWNrTk8
2dXhSNk44NUlGU21sUjFnWll1TytxU3ZyOFJNRjR5WmxQdW1nPT0iLCJtYWMiOiJmM2Y0ZDYxZDNlYTY1Z
DNiYzE4ZWVlNDUzYTFkMzc2NGZlYzU2MjE3MjgwYjRkZmU2NjUwMDIzODE2N2E5NDkyIn0%3D; expires
=Wed, 25-Jul-2018 11:44:47 GMT; Max-Age=7200; path=/; httponly
Content-Type: text/html; charset=UTF-8

~
```

Meta tag



Otherwise, browsers try to “guess” if these information are not present

Practical Considerations

- If you are working on web, use UTF-8
- If your operation is mostly with GUI and calling windows APIs with Unicode string, use UTF-16
- UTF-16 takes least space than UTF-8 and UTF-16 if most characters are Asian
- UTF-8 takes least space if most characters are Latin
- If memory is cheap and you need fastest operation, random access to characters etc, use UTF-32
- If dealing with Endianness & BOM is a problem, then use UTF-8
- When in doubt, use UTF-8 😛

https://avro.im/utf.pdf